A Comparison Between the PLM and the MC68020 as Prolog Processors

Yale N. Patt and Chien Chen

Computer Science Division
University of California, Berkeley.
Berkeley, CA 94720

1. Introduction.

1.1. Overview.

This study continues along the same vein as the work reported by Mulder and Tick [1], which compared the relative performance of the general purpose MC68020 available from Motorola and the special purpose PLM developed at U.C. Berkeley [2,3], with respect to the execution of Prolog programs. Both engines are based on the Warren Abstract Machine; the PLM directly executes WAM constructs and the MC68020 executes a 68020 image that was generated by macroexpanding each WAM construct.

The study reported here compares a more recent Prolog processor, the Berkeley VLSI-PLM with the Motorola MC68020 across fourteen Prolog benchmarks. Comparisons were made for 10 MHz and 16.7 MHz versions of the VLSI-PLM on the one hand, versus 10 MHz, 12 MHz, 16.7 MHz, 25 MHz, 30 MHz, and 40 MHz versions of the MC68020. The comparisons reflect the timing information in the MC68020 User's Manual [4] which identifies three levels of performance, depending on whether instructions can overlap their execution, whether instruction accesses hit in the small on-chip instruction cache, or neither.

Report Documentation Page	Form Approved OMB No. 0704-0188

1. REPORT DATE **JAN 1988**	2. REPORT TYPE	3. DATES COVERED **00-00-1988 to 00-00-1988**

4. TITLE AND SUBTITLE **A Comparison Between the PLM and the MC68020 as Prolog Processors**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Different execution models for Prolog have different cost/performance benefits and different performance opportunities. In this paper, we treat two implementations of Prolog, a tailored special purpose WAM processor, the Berkeley VLSI-PLM, and an off-the-shelf general purpose part, the MC68020. This work continues along the same vein as the work reported by Mulder and Tick. Fourteen Prolog benchmarks are compiled to WAM code, and their execution times on the VLSI-PLM and the MC68020 are compared and analyzed. Additional experiments are performed to calibrate the calculated MC68020 execution times with actual execution times of a SUN3/260 and a NCR Tower/32. The paper concludes with a number of observations.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**62**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

1.2. Objectives.

We have performed this study for several reasons. First, our long-term objectives are to understand the relative performance opportunities as well as the relative cost/performance benefits associated with various implementations of Prolog. Toward that end, a natural experiment is to compare a tailored special purpose Prolog processor to an off-the-shelf general purpose part.

Second, the recent study referred to above [1], which compared the Berkeley PLM and the general purpose MC68020, raised certain concerns which have compelled us to continue this comparison. For example, Mulder and Tick relied on a significance method for comparing performance. We felt that the additional work encountered in translating all 45 PLM constructs to MC68020 machine instructions, rather than the fourteen they translated, would produce more meaningful results. Also, their comparison involved only three benchmarks. We felt that the comparison would be improved if the benchmark set were expanded. While we still need to work on developing a proper set of benchmarks, we did expand the set used from three to fourteen. Most important, Mulder and Tick relied on their Lcode architecture model and their extensions to the Berkeley PLM compiler for obtaining measurements for the Berkeley PLM. In the case of the architecture model, there are substantial differences between the Lcode architecture and the PLM. In the case of the compiler, the techniques used in some instances were quite different. Since we have available the Berkeley PLM compiler [5] and the various Berkeley PLM simulators [6,7], we were able to more accurately simulate the benchmark set on our special purpose processors.

Finally, our research in Prolog microarchitecture has reached the point where we have designed our first single chip implementation, the VLSI-PLM, and it seems appropriate to compare its performance with that of a single chip general purpose processor.

1.3. Organization of this report.

This report is organized into five sections and two appendices. Section 2 describes the mapping between the requirements of the Warren Abstract Machine and its implementation with the MC68020. Our mapping is slightly different from the one given in [1]. Section 3 describes the methodology used to compare the two implementations, and reports the results of the comparison. Various problems endemic to this comparison method are discussed. In section 4, we analyze these results and offer a number of observations. In section 5, we summarize the results of this study. Appendix A contains the MC68020 machine language emulation for each of the 45 PLM instructions, along with the corresponding timing information. Appendix B contains the five tables which report the detailed comparison data described in Section 3.

2. Mapping the Warren Abstract Machine onto the MC68020.

Before we could execute WAM code on the MC68020, we needed to identify each of the abstract WAM instructions in the context of real MC68020 instructions. We performed this mapping with the goal of making the MC68020 an effective Prolog processor, removing unnecessary bottlenecks where they presented themselves.

Figure 1 shows the mapping of the WAM registers into the MC68020. Since the number of programmer visible registers in the PLM is more than that available in the MC68020, we were forced to put some PLM registers into memory for the MC68020. Because the MC68020 requires at least three cycles for a memory access, we tried to assign the least frequently used registers to memory. We also eliminated some PLM registers by slightly changing the WAM definition, if this were advantageous to the MC68020. These changes are listed below in detail.

1. We eliminated the HB register, since the net effect of keeping it in memory would have been the same as accessing it through the B register, which points to the choice point frame containing HB.

2. According to both Dobry [6] and Mulder and Tick [1], the gain obtained from environment trimming is negligible. The execution model in [1] does not implement

it; we decided not to implement it either.† In order to achieve this, we changed the semantics of the *allocate* instruction so that it has an argument indicating how much space is to be allocated for the environment. In addition, this eliminated the need for the N register which contained the number of permanent variables needed after returning from a call. We also changed the addressing mechanism for the permanent variables to a negative offset from the environment pointer instead of a positive offset. The semantics of *allocate N* require that the top of the stack be incremented by N with the top four locations dedicated for E, CP, B and cut-flag. Permanent variables are located below these four locations. Note that the variable number starts at Y4 instead of Y0.

3. We assigned separate *unify* instructions for read and write modes. This eliminated the need for a register and a comparison at run time to test the mode. This technique is based on the fact that once unification proceeds in the write mode, the following unifications are all in the write mode. This required some changes in the compiler, but we feel the changes are minor and will not result in a significantly larger code size.

4 We used the two least significant bits in a data word as the primary tag to distinguish among reference, constant, list and structure; see figure 2. The next two least significant bits are used as the secondary tag to distinguish between integer, atom, and other data types. Note we have used a tagging scheme different from the one used for the PLM in order to speed up tests for integers and dereferencing. We used the most significant bit as the *cdr* bit for cdr-compressed list representation.

† By eliminating environment trimming, we improve the performance of the MC68020. However, this improvement represents a savings of fewer than 1% of the total cycles needed to execute the benchmarks.

3. Experimental Methodology.

3.1. Basic Measurements.

Our objective was to compare the performance of a special purpose processor (the VLSI-PLM) with that of a general purpose processor (MC68020) on a set of Prolog benchmarks. First, using our Prolog compiler [5], we compiled all benchmarks into modified WAM (i.e., PLM) code. For the MC68020, we then macro-expanded all PLM instructions and escapes into MC68020 machine code. We attempted only limited optimization for the resulting 68020 instructions. For the VLSI-PLM, PLM instructions execute directly in microcode, while those escapes needed by the benchmarks (with the exception of mod and div) generate call instructions to library routines which themselves are made up of VLSI-PLM instructions.

We used the VLSI-PLM simulator [7,8] to obtain the frequencies of each PLM instruction, as well as the frequencies of failure, dereference, trail, and decdr. For each PLM instruction, we also obtained the frequencies of different execution paths.

We used the *Motorola User's Manual* [4] to generate a timing table consisting of the number of MC68020 cycles required to execute each PLM instruction, according to best, cache and worst case situations for executing each MC68020 instruction. These three situations correspond, respectively, to whether both MC68020 instruction overlap is possible and the instruction access hits in the 256 byte on-chip instruction cache, whether instruction overlap is not possible but the instruction access hits in the on-chip cache, or whether neither is possible. Appendix A contains this cycle count information. Since the execution time of some PLM instructions is very data dependent, we also included separate entries for each subcase.

To calculate the number of cycles the MC68020 would take to execute each benchmark program, we multiplied the occurrence of each PLM instruction by its corresponding entry in the timing table. The number of cycles the VLSI-PLM would take to execute each benchmark was obtained by running the benchmark on the VLSI-PLM simulator.

Table 1 (see Appendix B) reports the ratio of machine cycles required by the MC68020 and by the PLM to execute each of the fourteen benchmarks. Three entries are reported for each benchmark, reflecting the best case, cache case, and worst case described above. We also included in Table 1, for comparison purposes, the cycle counts for the three benchmarks reported in [1].

To calculate execution times for the VLSI-PLM and for the MC68020 chip, we assumed cycle times as follows: For the VLSI-PLM, we assumed two clock rates, 10 MHz and 16.7 Mhz. For the MC68020, we assumed clock rates of 10 Mhz, 16.7 Mhz, 25 Mhz, 30 Mhz and 40 Mhz. We assumed that the memory system could respond within one cycle for the PLM and within three cycles for the MC68020, since the MC68020 requires at least three cycles for a memory access.

Table 2 shows the relative performance (i.e., the reciprocal of execution time) of the various frequency MC68020s, normalized to the 10 MHz PLM. Table 3 shows the equivalent relative performance of the various frequency MC68020s, normalized to the 16.7 MHz PLM. Table 3 has been included in this report, because our current understanding of microarchitecture for Prolog (c.f., section 4.1) makes a 16.7 MHz PLM not a difficult challenge.

3.2. Calibration.

The calculations reported in Tables 2 and 3 describe very different levels of performance for the MC68020, depending on which of the three cases (best, cache, or worst) we choose to believe is most nearly correct. To calibrate our calculated data, we ran several of the benchmarks on a SUN 3/260, operating essentially stand-alone, with a very large cache. Assuming a 100% cache hit ratio, and no wasted cycles due to memory delays, the measured data on the SUN 3/260 should correspond very nearly to the calculated results. Table 4 reports the calculated cycles vs. the real cycles for six of the benchmarks. It appears that the real execution time is somewhere between the cache case and the worst case.

4. Analysis of the Results, and some Observations.

4.1. Microarchitectures for Prolog.

The VLSI-PLM represents one implementation in a sequence of designs, not the end result. That is, although the results of Tables 2 and 3 are quite respectable, it is important to keep in mind that the VLSI-PLM in no way sets a limit on the performance that can be obtained with special purpose Prolog processors.

Our first implementation, the Berkeley PLM, was designed in 1983 and 1984 [2,3]. As our work has progressed, our understanding of Prolog has increased. This increased understanding is reflected in the VLSI-PLM, for example, with respect to the execution of built-ins, that is, as library routines consisting of instructions from the base PLM machine. This technique, sometimes referred to as "millicode," is the mechanism Digital Equipment Corporation used to implement the full VAX architecture on the microVAX II chip.

Furthermore, already in the pipe is one of our current designs, PUP (parallel unification of Prolog), which achieves (based on a full register transfer level simulator written in N2) about twice the performance of the VLSI-PLM by concurrently processing WAM instructions by means of multiple function units [9].

Finally, two other things should improve the performance of a single chip Prolog processor relative to a general purpose MC68020: tuning the VLSI circuitry and tuning the microarchitecture. With respect to circuits, as advanced VLSI technology becomes more readily available, the wide disparity between the degree to which one can tune a special purpose circuit and the degree to which one can tune a high performance general purpose part should diminish. With respect to microarchitecture, the VLSI-PLM has not yet been aggressively pipelined. Continued attention to critical path design should produce an improved cycle time. The 16.7 MHz clock suggested in Table 3 is a reflection of that.

4.2. Degradation due to unoptimized MC68020 code.

We were concerned, in fairness to advocates of off-the-shelf hardware, coupled usually with a reliance on optimizing compilers, that our MC68020 target machine code does in fact suffer from a fairly pedestrian hand-compilation from Prolog via WAM intermediate code. We see a developing industry, represented by Quintus and BIM, for example, which suggests that optimal execution of Prolog on off-the-shelf processors will have the advantage of heavily optimized compilations.

To measure the degradation due to our straightforward macro expansion, we tested six of the benchmarks on our Tower workstation (a 16.7 MHz 68020). We compared the number of cycles required by our macroexpansion method with the number of cycles required by the Quintus system (release 1.5, running under UTS V). Table 4 shows the number of cycles required to execute each of six Prolog benchmarks by the two methods. Over the six benchmarks, the Quintus system outperformed the macroexpanded version on five of them. We are not drawing any strong conclusions on this little data other than to say that it appears our macroexpanded code does not seriously skew our comparisons.

4.3. Code Explosion.

Another concern that should not be minimized in any comparative study of special purpose vs. general purpose processors is the code explosion problem. In most cases, we have found that compiling to a lower level architectural interface results in a significant increase in the code size [10]. Table 5 shows the relative code sizes of the PLM and the macro expanded MC68020 over eight of the 14 benchmarks. This code explosion, more than 15 to 1 in many cases, does degrade performance with respect to memory bandwidth and cache hit ratio. The simplistic answer of a larger cache is unacceptable since it is a fact that larger caches are slower caches. Although the benchmarks of this study fit well within the limits of the cache, Prolog applications of the future will not be able to.

4.4. The Significance Method used by Mulder and Tick.

We were concerned that the significance method used by Mulder and Tick could have created the appearance of convergence, when in reality, the "next" non-implemented PLM construct would have caused divergence. This certainly could have been the case. It turned out, however, that the results obtained by carrying out the macroexpansions for all 45 PLM constructs mirrored those obtained by stopping after the first 14. The first entries in Table 1 show the results for the benchmarks chat and boyer obtained via the significance method (Mulder-Tick) and via macroexpanding all 45 constructs (Patt-Chen). The difference in the numbers reported in Table 1 are more likely to be due to differences between the two exection models (i.e., plm1 vs. VLSI-PLM), rather than due to any differences resulting from the significance method.

4.5. Best, Cache, and Worst Cases.

The *Motorola User's Manual* gives timing information for the following situations. The *best* case is when the instruction is in the on-chip instruction cache and maximum overlap is achieved. The *cache* case is when the instruction is in the cache but there is minimum overlap between instructions. The *worst* case is when the instruction is not in the cache and there is no overlap between instructions. We summed all cycles according to the three cases. For the best case, we can safely conclude that it is a very optimistic upper bound on performance, not something that is really achievable. For example, the code sequence to macroexpand the switch_on_tag routine (reproduced as figure 3) which forms the core of the switch instructions yields a best case of eight cycles. However, a small fine-tuned measurement demonstrated that it can not execute in less than 28 cycles.

More importantly, recall the data of Table 4. We conclude that the real performance of the MC68020 is probably somewhere between the cache case and the worst case, rather than close to the best case.

4.6. Memory Cycle Differences.

The basic memory access protocols for the MC68020 and the PLM are different. The PLM is designed with a cache system and a write buffer, enabling the memory system to respond in one cycle. The MC68020, on the other hand, uses at least 3 cycles for a memory access. This difference between memory access protocols for the MC68020 and the PLM is very important to the relative performance of the two machines for the following reason. First, since the MC68020 code size is much greater, and its on-chip cache consists of only 256 bytes, it needs to make a lot more instruction fetches than the PLM. Also, like most AI software, Prolog programs tend to be memory intensive (i.e., more than three data accesses per PLM instruction is not uncommon). Both situations result in the MC68020 spending much more time in memory accesses due to its three-cycle protocol.

4.7. Cdr-compressed List Representation.

The use of cdr–compressed list representation has received substantial treatment in the Prolog literature. The VLSI-PLM uses it and has hardware support for checking the cdr bit. The MC68020 has no such hardware support, and so incurs a penalty each time it needs to check the cdr bit. Dobry [2] observed that lists tend to be non-contiguous after some manipulation. For such lists, the VLSI-PLM suffers little penalty compared with the cdr–compressed case. One can argue legitimately that the use of cdr–compressed lists unfairly penalizes the MC68020. On the other hand, cdr–compressed lists save memory accesses, and as we have already discussed, this is a major bottleneck in MC68020 performance. On balance, the use of cdr–compressed lists probably hurts the MC68020, although how much so is not clear. The extra memory access is coupled with simpler (smaller in size and fewer branches) instructions. In this study, we elected to use the cdr–compressed list representation to maintain consistency across the Prolog benchmarks, although we recognize that by so doing, we may have penalized the MC68020 unfairly – estimated to be as much as 10% in the case of the chat benchmark, around 5% in the case of boyer.

4.8. Hardware Support for Tags.

The VLSI-PLM has an advantage over the MC68020 for instructions involving tag manipulation, for example, dereference and general unification. This is mainly due to three factors: (1) The PLM has a powerful multiway branch mechanism based on tags. (2) The PLM can do tag manipulation on programmer invisible registers. (3) The MC68020 suffers from the separation of its address and data registers, i.e., the data registers can not be used to address memory directly and the address registers can not be used for tag manipulation.

However, the PLM does not have as great an advantage for instructions which do not require tag manipulation. These instructions include those which manipulate choice points, put instructions, and write-mode unification instructions. Unfortunately for the PLM, these instructions occur almost as frequently as those that require tag manipulation. For example, for the relational operator built-ins, when the PLM has to go off-chip and it can not utilize its tag manipulation capability, the VLSI-PLM needs 54 cycles while the MC68020 needs 17 cycles in the best case, 78 cycles in the cache case, and 99 cycles in the worst case. At best, this produces a cycle count advantage for the VLSI-PLM of 1.8 to 1.

5. Concluding Remarks.

This report has attempted to continue along the same vein as the work of Mulder and Tick [1] and further compare the performance of a special purpose processor and a general purpose processor with respect to the execution of Prolog benchmarks. For the general purpose processor, we continued with Mulder and Tick's choice of the MC68020. For the special purpose processor, we chose our most recent implementation, the VLSI-PLM, which is currently in fabrication.

A fair comparison is fraught with obstacles. There is no existing VLSI-PLM system, so we can not simply run the benchmarks on both systems. On the other hand, we do not have a comprehensive MC68020 simulator so we can not simply count sanitized cycles in both cases.

Our method was to first compile Prolog benchmarks to modified WAM code. After that, in one case we executed the WAM code on the VLSI-PLM simulator provided by [7], and in the other case, macroexpanded the WAM code into MC68020 code and counted the number of cycles it would take to execute. To calibrate the 68020 calculations, six of the 14 benchmarks were executed on a very lightly loaded SUN 3/260 system containing enough cache memory (64 KB) to reasonably guarantee there would be no wait states waiting for memory.

Although several problems with this study exist, some general statements are possible as delineated in section 4. The most relevant performance figures are those contained in Table 3, which assumes a 16.7 MHz VLSI-PLM. If we assume approximately worst case 68020 behavior, suggested by Table 4, and a 30 MHz 68020, we see about a factor of between three and four in performance in favor of the VLSI-PLM. If we add to this an improved VLSI technology implementation and the effects of significant code explosion (15 to 1 not uncommon, see Table 5), the performance ratio can be even greater.

Acknowledgement.

This report, like most of the work done on the Aquarius project, draws on the knowledge and insights of many members of our research team, rather than simply those of the authors of this report. It is our pleasure to gratefully acknowledge the help of Bruce Holmer, Vason Srini, and Alvin Despain. Financial support to carry out this work came from a number of sources, including NCR Corporation, Digital Equipment Corporation, and the California Micro program. Part of this work was sponsored by Defense Advanced Rsearch Projects Agency (DoD), Arpa Order No. 4871, Monitored by Space & Naval Systems Warfare Command under Contract No. N00039-84-C-0089.

References.

[1] Mulder, H., Tick, E. *A performance Comparison between PLM and an MC68020 Prolog Processor*, Technical Note No. CSL-86-302. (Sept. 1986).

[2] Dobry, T., Patt, Y. and Despain, A., Design Decisions Influencing the Microarchitecture for a Prolog Machine, *Conf. Proc., 17th Annual International Workshop on Microprogramming,* (October 1984).

[3] Dobry, T., Despain, A. and Patt, Y., Performance Studies of a Prolog Machine Architecture, *Conf. Proc., 12th Annual International Symposium on Computer Architecture, pp. 180-190,* (June 1985).

[4] Motorola. *MC68020, 32-Bit Microprocessor User's Manual.*

[5] VanRoy, P., *A Prolog Compiler for the PLM,* Master Thesis, U.C Berkeley. (August, 1984)

[6] Dobry, T. *A High Performance Architecture for Prolog,* Ph. D dissertation, Report No. UCB/CSD 87/352. (May 1987).

[7] Holmer, B. *VLSI-PLM Simulator and User Manual,* in preparation.

[8] Srini, V.P., Tam, J.V., Nguyen, T.M., Patt, Y.N., Despain, A.M., Moll, M., and Ellsworth, D., A CMOS Chip for a Prolog Processor, *Proceedings of ICCD 1987,* New York, (October, 1987).

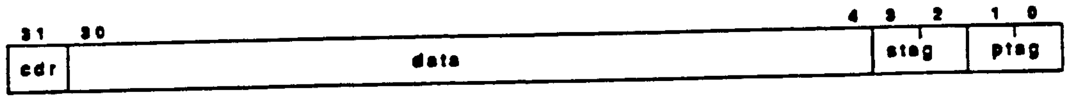
[9] Chen, C., Singhal, A. and Patt, Y., *PUP: An Architecture to Exploit Parallel Unification in Prolog,* unpublished report (October, 1987).

[10] Patt, Y., Several Implementations of Prolog, *IEEE Transactions on Systems, Man, Cybernetics,* (accepted for publication, Spring 1988).

Figure 1. Register Mapping between PLM and MC68020

MC68020	Function	PLM
A2	Choice Point Pointer	B
A3	Environment Pointer	E
A4	Heap Pointer	H
A5	Structure Pointer	S
A6 (MP)	Memory State Overflow Pointer	No PLM equivalence
A7	Push-down List Pointer	PDL
D2-D7	Argument Register 0 to 5	A0-A5
D2-D7	Temporary Register 0 to 5	X0-X5
Memory Yn_offset(E)	Permanent Variable n	Yn
Memory TR_offset(MP)	Top of Trail Stack	TR
Memory CP_offset(MP)	Continuation Pointer	CP
Memory X6_offset(MP)	Argument Register 6	A6
Memory X7_offset(MP)	Argument Register 7	A7
Memory X6_offset(MP)	Temporary Register 6	A6
Memory X7_offset(MP)	Temporary Register 7	A7
memory CUT_offset(MP)	cut-flag	cut in PLM
memory H2_offset(MP)	For set/access	H2
D0,D1	Scratch Data Registers	No PLM Equivalence
A0,A1	Scratch Address Registers	No PLM Equivalence

Figure 2. Data Encoding Format for MC68020



ptag	00	reference type data and stag together form a 29 bit pointer
	01	list type data and stag together form a 29 bit pointer
	10	constant type data is interpreted according to stag
	11	structure type data and stag together form a 29 bit pointer
stag	00	the data is an integer
	10	the data is an atom
	11	the data is neither an integer nor an atom
cdr	0	the data is cdr-compressed
	1	the data is not cdr-compressed, data and the stag form a 29 bit pointer

Figure 3. switch_tag Macro and Timing

switch_tag Dx				
input:	Dx, Jumplist			
output:				
function:	Dx should be dereferenced. goto appropriate address according to datatype of Dx			
Instruction		best	cache	worst
switch_tags:				
movea	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
and.w	#3,Dx	0+0 (0/0/0)	2+2 (0/0/0)	3+3 (0/2/0)
exg	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
movea	Jumplist(PC,A0), A0	4 (1/0/0)	9 (1/0/0)	11 (1/2/0)
jmp	(Jumplist,PC,A0)	1+3 (0/0/0)	4+6 (0/0/0)	7+6 (0/2/0)
Jumplist:				
_$var	(00)			
_$list	(01)			
_$const	(10)			
_$struct	(11)			
cycles		8 (0)	27 (0)	36 (8)

Appendix A. Macroexpanded WAM Constructs and Timing Information.

A.1 Macro Expansions

trail Dx,Ax

input:	Dx - address to be trailed with cdr-bit Ax - address to be trailed with cdr-bit cleared
output:	
temp:	A0
function:	Dx may have cdr bit set, Both Dx and Ax have address to be trailed

Instruction		best	cache	worst
trail:				
exg	Dx,Ax	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
cmp.l	#stackbase,Dx	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bge	1	1/3	4/6	5/9
cmp.l	B,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	4	1/3	4/6	5/9
bra	2	3	6	9
1:				
cmp.l	HOFFSET(B),Dx	0+ 3 (1/0/0)	2+ 5 (1/0/0)	3+ 6 (1/2/0)
blt	4	1/3	4/6	5/9
2:				
exg	Ax,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
movea.l	TR(MP),A0	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
move.l	Dx,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
subq	#4,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	A0,TR(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
4:				
cycles	Dx < HB	16 (4)	45 (4)	61 (18)
	HB < Dx < stackbase	9 (1)	27 (1)	38 (10)
	stackbase < Dx < B	14 (3)	44 (3)	60 (17)
	B < Dx	4 (0)	20 (0)	28 (7)

dereference Dx				
input:	Dx, $Lref, $Lnref			
output:	Dx			
temps:	A0			
function:	dereference and branch according to result			
Instruction				
dereference:				
bftst	Dx {0,2}	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
bne	$Lnref	1/3	4/6	5/9
1:				
bclr	#cdr,Dx	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
2:				
movea.l	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(A0),Dx	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bftst	Dx {0,2}	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
bne	$Lnref	1/3	4/6	5/9
bclr	#cdr,Dx	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bne	3	1/3	4/6	5/9
cmp.l	A0,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
beq	$Lref	1/3	4/6	5/9
bra	2	3	6	9
3:				
cmp.l	A0,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	2	1/3	4/6	5/9
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	$Lref	3	6	9
cycles	Dx nref	6 (0)	12 (0)	16 (3)
	1 level to nref	14 (1)	34 (1)	43 (9)
	Dx ref	5 (0)	12 (0)	17 (5)
	each level	12 (1)	34 (1)	44 (10)
	each cdr	2 (0)	2 (0)	4 (1)

switch_tag Dx				
input:	Dx, Jumplist			
output:				
function:	Dx should be dereferenced goto appropriate address according to datatype of Dx			
Instruction		best	cache	worst
switch_tags:				
movea	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
and.w	#3,Dx	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
exg	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
movea	Jumplist(PC,A0), A0	4 (1/0/0)	9 (1/0/0)	11 (1/2/0)
jmp	(Jumplist,PC,A0)	1+ 3 (0/0/0)	4+ 6 (0/0/0)	7+ 6 (0/2/0)
Jumplist:				
_$var	(00)			
_$list	(01)			
_$const	(10)			
_$struct	(11)			
cycles		8 (0)	27 (0)	36 (6)

decdr

input:	Dx,Ax $Lcdr_ref, $Lcdr_nref, $Lothers
output:	Dx,Ax
function:	Trace cdr pointers goto labels accordingly

Instruction		best	cache	worst
decdr				
bftst	Dx [0,2]	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
bmi	$Lcdr_nref	1/3	4/6	5/9
1:				
bclr	#0,Dx	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
2:				
movea.l	Dx,Ax	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(Ax),Dx	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	4	1/3	4/6	5/9
bftst	Dx [0,2]	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
bgt	1	1/3	4/6	5/9
bmi	$Lcdr_nref	1/3	4/6	5/9
bclr	#cdr, Dx	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
cmp.l	Ax,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	2	1/3	4/6	5/9
bra	$Lcdr_ref	3	6	9
4:				
addq.l	#4,Ax	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	$Lothers	3	6	9
$Lcdr_ref:				
$Lothers:				
$Lcdr_nref				
cycles	to $Lothers	2 (0)	8 (0)	12 (4)
	to $Lcdr_ref	5 (0)	28 (0)	35 (6)
	to $Lcdr_nref	6 (0)	14 (0)	17 (3)
	each level	12 (1)	28 (1)	36 (8)

Utilities to test data type

input	Dx, $Lfalse
output:	
function:	Test for the data type if true, fall through, otherwise, brance to $Lfalse

Instruction		best	cache	worst
test_reference				
bftst	Dx {0,2}	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
bne	$Lfalse	1/3	4/6	5/9
cycles	true	4 (0)	10 (0)	12 (2)
	false	6 (0)	12 (0)	16 (3)
test_list				
bftst	Dx{0,2}	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
ble	$Lfalse	1/3	4/6	5/9
cycles	true	4 (0)	10 (0)	12 (2)
	false	6 (0)	12 (0)	16 (3)
test_const				
movea	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
and.b	#3,Dx	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
cmp.b	#2,Dx	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
exg	A0,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	$Lfalse	1/3	4/6	5/9
cycles	true	1 (0)	16 (0)	23 (7)
	false	3 (0)	18 (0)	27 (8)
test_struct				
movea	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
add.b	#3,Dx	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
cmp.b	#3,Dx	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
exg	A0,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	$Lfalse	1/3	4/6	5/9
cycles	true	1 (0)	16 (0)	23 (7)
	false	3 (0)	18 (0)	27 (8)
test_int				
movea	Dx,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
and.b	#7,Dx	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
cmp.b	#2,Dx	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
exg	A0,Dx	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	$Lfalse	1/3	4/6	5/9
cycles	true	1 (0)	16 (0)	23 (7)
	false	3 (0)	18 (0)	27 (8)

A.2. Get Instructions

get_constant_X				
input:	constant C and argument register number Xi			
output:				
function:	Unify C with Xi			
Instruction		best	cache	worst
get_constant_X:				
move.l	#C,D0	0 (0/0/0)	6 (0/0/0)	5 (0/1/0)
move.l	Xi,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	_unify1	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
cycles		8+ u-4 (2)	20+ u-10 (2)	33+ u-12 (7)
move.l	Xi(MP),D1	3 (1/0/0)	7 (1/0/0)	9 (1/1/0)
	Xi in memory	3 (1) more	5 (1) more	6 (2) more

get_value_X				
input:	input arguments Xi and Xj			
output:				
function:	unify Xi with Xj			
Instruction		best	cache	worst
get_value_X:				
move.l	Xi,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	Xj,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	_unify	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
cycles		8+ u • (2)	16+ u • (2)	26+ u • (7)
	X in mem	3 (1) or 6 (2)	7 (1) or 14 (2)	9 (1) or 18 (2)
• u is the time spent in the unification routine				

get_value_Y				
input:	Permenant variable Yi and argument Xj			
output:				
function:	unify Yi and Xj			
Instruction		best	cache	worst
get_value_Y:				
move.l	-4•i(E),D0	3 (1/0/0)	7 (1/0/0)	9 (1/1/0)
move.l	Xj,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	PDL,pldBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	_unify	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
cycles		11+ u • (3)	21+ u • (3)	32+ u • (8)
	Xj in memory	3 (1) more	7 (1) more	9 (1) more
• u is the time needed by the unification subroutine.				

get_variable_X				
input:	argument registers Xi and Xj			
output:				
function:	Move the content of Xj to Xi			
Instruction		best	cache	worst
get_variable_X:				
move.l	Xj,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
cycles		0 (0)	2 (0)	3 (1)

get_variable_X				
input:	argument registers Xi and Xj			
output:				
function:	Xj is in memory rather than a machine register			
Instruction		best	cache	worst
get_variable_X:				
move.l	Xj,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
cycles		3 (1)	5 (1)	7 (2)

get_variable_X				
input:	argument registers Xi and Xj			
output:				
function:	Xj is in the memory			
Instruction		best	cache	worst
get_variable_X:				
move.l	Xj(MP),Xi	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
cycles		3 (1)	7 (1)	9 (3)

get_variable_X				
input:	argument registers Xi and Xj			
output:				
function:	Both Xi and Xj are in memory			
Instruction		best	cache	worst
get_variable_X:				
move.l	Xj(MP),Xi(MP)	6 (1/0/1)	8 (1/0/1)	13 (1/2/1)
cycles		6 (2)	8 (2)	13 (4)
c s s s s.				
get_variable_Y				
input:	Permenant variable Yi and argument register Xj			
output:				
function:	move the content of Xj into Yi			
get_variable_Y:				
move.l	Xj,-4*i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
cycles	Xj in register	3 (1)	5 (1)	7 (2)

get_variable_Y				
input:	Permenant variable Yi and argument register Xj			
output:				
function:	Xj is in memory			
Instruction		best	cache	worst
get_variable_Y:				
move.l	Xj(MP),-4*i(E)	6 (1/0/1)	8 (1/0/1)	13 (1/2/1)
cycles		6 (2)	8 (2)	13 (4)

get_list				
input:	argument regiser Xi			
output:				
function:	set up for unification of Xi and a list			
Instruction		best	cache	worst
get_list Xi:				
dereference	(Xi, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
move.l	H,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.b	#listtag,D0	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
btst	#cdr,Xi	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	1	1/3	4/6	5/9
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
1:				
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	Xi,A0	4 (0)	20 (0)	28 (7)
move.l	D0,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	write_unify	3	6	9
Lnref:		6 (0)	12 (0)	16 (3)
move.l	Dx,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bftst	D0 [0,2]	3 (0/0/0)	6 (0/0/0)	7 (0/1/0)
ble	_fail	1/3	4/6	5/9
bclr	#0,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D0,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
read_unify_				
cycles	reference	19+ d+ t (1)	62+ d+ t (1)	86+ d+ t (23) *
	list	11+ d (0)	30+ d (0)	39+ d (8) *
	others	12+ d (0)	26+ d (0)	32+ d (7) *
	Xi in memory	3 (1) more	9 (1) more	12 more (4)

dereference and *trail* are to be macroexpanded

* d is time needed to derefence and t is the time needed to do trail checking

get_structure

input:	Xi
output:	
function:	set up for the unification of Xi and a structure

Instruction		best	cache	worst
get_structure:				
dereference	(Xi, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
move.l	H,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.b	#structtag,D0	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
btst	#cdr,Xi	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	1	1/3	4/6	5/9
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
1:				
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	Xi,A0	4 (0)	20 (0)	28 (7)
move.l	D0,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	#Functor,(H)+	4 (0/0/1)	8 (0/0/1)	7 (0/1/1)
bra	write_unify_	3	6	9
Lnref:				
test_struct	(D0, _fail)	1/3 (0)	16/18 (0)	23/27 (7)
and.b	#0xfc,D0	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
movea.l	D0,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
cmpi.l	#F,(S)+	0+ 5 (1/0/0)	2+ 8 (1/0/0)	3+ 9 (1/1/0)
bne	_fail	1/3	4/6	5/9
ready_unify_				
cycles •	reference	23+ d+ t (2)	70+ d+ t (2)	93+ d+ t (25)
	structure	13+ d (1)	48+ d (1)	65+ d (16)
	list or const	9+ d (0)	30+ d (0)	43+ d (11)
	functor not match	15+ d (1)	50+ d (1)	69+ d (17)
	Xi in memory	3 (1) more	7 (1) more	9 (4) more

dereference, *trail* and *test_struct* are to be microexpanded
• d time needed for dereference, and t time needed for trail the bindings

A.3. Put Instructions

put_variable_X				
input:	argument register Xi			
output:				
function:	Create an unbound variable on the heap and put the pointer in Xi			
Instruction		best	cache	worst
put_variable_X:				
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
cycles		4 (1)	6 (1)	8 (3)

put_variable_Y				
input:	Permenant variable Yi			
output:				
function:	Create an unbound permenant variable Yi			
Instruction		best	cache	worst
put_variable_Y:				
lea	-4•i(E),A0	4 (0/0/0)	4 (0/0/0)	6 (0/2/0)
move.l	A0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
move.l	A0,Xj	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
cycles	Xi in register	7 (1)	10 (1)	14 (5)

put_value_X				
input:	argument Xi and Xj			
output:				
function:	move the content Xi into Xj			
Instruction		best	cache	worst
put_value_X:				
move.l	Xi,Xj	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
cycles	Xi,Xj in registers	0 (0)	2 (0)	3 (1)
	Xi in memory	3 (1)	7 (1)	9 (2)
	Xj in memory	3 (1)	5 (1)	7 (2)
	both Xi,Xj in memory	6 (2)	8 (2)	13 (3)
This instruction is the same as the get_value_X, except the direction of data movement is reversed				

put_value_Y				
input:	permenant variable Yi and argument register Xj			
output:				
function:	Move the content of Yi into Xj			
Instruction		best	cache	worst
put_value_Y:				
move.l	-4•i(E),Xj	3 (1/0/0)	7 (1/0/0)	9 (1/1/0)
cycles	Xj in register	3 (1)	7 (1)	9 (2)
	Xj in memory	6 (2)	8 (2)	13 (3)

put_constant Xi				
input:	Constant C and argument register Xi			
output:				
function:	move C into Xi			
Instruction		best	cache	worst
put_constant:				
move.l	#Constant,Xi	0 (0/0/0)	6 (0/0/0)	5 (0/1/0)
cycles	Xi in register	0 (0)	6 (0)	5 (1)
	Xi in memory	3 (1)	9 (1)	9 (2)

put_list				
input:	argument Xi			
output:				
function:	initialize Xi to point to the heap where the list is built			
Instruction		best	cache	worst
put_list:				
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.b	#listtag,Xi	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
write_unify_:				
cycles	Xi in register	0 (0)	6 (0)	9 (3)
	Xi in memory	3 (1)	11 (1)	16 (4)

put_structure				
input:	Functor F and argument Xi			
output:				
function:	initialize Xi to point to the heap where a structure with functor F is built			
Instruction		best	cache	worst
put_structure:				
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#structtag,Xi	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
move.l	#F,(H)+	4 (0/0/1)	8 (0/0/1)	7 (0/1/1)
write_unify_				
cycles	Xi in register	4 (1)	14 (1)	16 (5)
	Xi in memory	7 (2)	19 (2)	23 (7)

put_unsafe_value				
input:	Permenant variable Yi and argument register Xj (on chip)			
output:				
function:	If Yi is dereferenced to an unbound variable, a new variable is craeted on the heap and is put into Xj. Otherwise, the dereferenced value of Yi is put into Xj.			
Instruction		best	cache	worst
put_unsafe_value:				
move.l	-4*i(E),Xi	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
dereference	(Xi, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,Xi	0 (1/0/0)	2+ 4 (1/0/0)	3+ 5 (1/2/0)
bgt	Lnref	1/3	4/6	5/9
trail	Xi,A0	4 (0)	20 (0)	28 (7)
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
Lnref:		6 (0)	12 (0)	16 (3)
cycles	Xi reg. not ref	9+ d (1)	19+ d (1)	25+ d (6)
	ref no move	11+ d (2)	31+ d (2)	43+ d (13)
	ref move	17+ d (3)	55+ d (3)	75+ d (22)
Xi is in a machine register on chip				

put_unsafe_value				
input:	Permenant variable Yi and argument register Xj (in memory)			
output:				
function:	If Yi is dereferenced to an unbound variable, a new variable is craeted on the heap and is put into Xj. Otherwise, the dereferenced value of Yi is put into Xj.			
Instruction		best	cache	worst
put_unsafe_value:				
move.l	-4*i(E),D0	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
dereference	(D0, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,D0	0 (1/0/0)	2+ 4 (1/0/0)	3+ 5 (1/2/0)
bgt	Lnref	1/3	4/6	5/9
trail	D0,A0	4 (0)	20 (0)	28 (7)
move.l	H,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
Lnref:		6 (0)	12 (0)	16 (3)
move.l	D0,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
cycles	Xi in mem nref	12+ d (2)	24+ d (2)	32+ d (8)
	ref no move	14+ d (3)	36+ d (3)	50+ d (15)
	ref move	20+ d (4)	60+ d (4)	82+ d (24)

A.4. General Unification Routine

unify	
input:	D0, D1
output:	
temps:	D0,D1,S
function:	general unification subroutine

Instruction		best	cache	worst
dereference	(D0, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
dereference	(D1, Lrr, Lrnr)			
unify_rr:				
Lrr:		5 (0)	12 (0)	17 (5)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	10	1/3	4/6	5/9
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	20	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	2	1/3	4/6	5/9
1:				
movea.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
2:				
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	D1,A0	4+ t (0)	20+ t (0)	28+ t (7)
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
10:				
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	15	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	2	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
11:				
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	2	1/3	4/6	5/9
15:				
cmp.l	D0,D1	0 (0/0/0)	4 (0/0/0)	5 (0/1/0)
blt	11	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	11	3	6	9
20:				
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
ble	11	1/3	4/6	5/9
bra	1	3	6	9

unify (cont.)

Instruction		best	cache	worst
Lrar:		6 (0)	12 (0)	16 (3)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	21	1/3	4/6	5/9
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	D1,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	5 (0/1/0)
bra	2	3	6	9
21:				
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	1	3	6	9
unify1:				
Lnref:		6 (0)	12 (0)	16 (3)
dereference	(D1, Lnrr, Lnn)			
Lnrr:		5 (0)	12 (0)	17 (5)
btst	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	22	1/3	4/6	5/9
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	2	3	6	9
22:				
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	2	3	6	9
Lnn:		6 (0)	12 (0)	16 (3)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bclr	#cdr,D1 1 (0/0/0)	4 (0/0/0)	5 (0/1/0)	
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	30	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
unify_nn:				
30:				
switch_tag	(D0, Lfail, Ll, Lfail, Ls)	8 (0)	27 (0)	36 (6)
Ll:				
test_list	(D1, Lfail)	4/6	10/12	12/16
bra	unify_list	3	6	9
Ls:				
test_struct	(D1, Lfail)	1/3	16/18	23/27
bra	unify_struct	3	6	9
cycles	var,var	40+ d+ t (2)	96+ d+ t (2)	134+ d+ t (35)
	var,nvar	37+ d+ t (3)	88+ d+ t (3)	120+ d+ t (30)
	nvar,var	38+ d+ t (3)	88+ d+ t (3)	111+ d+ t (28)
	const,const	24+ d (1)	48+ d (1)	62+ d (11)
	list,list	32+ d+ ul (1)	83+ d+ ul (1)	95+ d+ ul (19)
	struc,struc	29+ d+ us (1)	89+ d+ us (1)	106+ d+ us (24)
	entry from unify1	6 (0) less	12 (0) less	16 (3) less

d time needed for dereference
t time needed for trail
ul and us are time needed to unify the list and structure respectively

unify_list

input:	D0, D1
output:	
temps:	D0,D1,S,A1,A0,PDL
function:	loop to unify lists

Instruction		best	cache	worst
_unify_list:				
move.l	(PDL)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
move.l	(PDL)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
unify_list:				
add.b	0xfc,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/1/0)
add.b	0xfc,D1	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/1/0)
movea.l	D0,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
movea.l	D1,A1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
move.l	(A1)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bra	_loop	3	6	9
_continue:				
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bmi	4	1/3	4/6	5/9
1:		2(0)•	8 (0)•	12 (4)•
move.l	(A1)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	_loop	1/3	4/6	5/9
decdr	(D1,A1,Lcr,Lfail,_loop)			
Lcr:		5 (0)	28 (0)	35 (6)
subq.l	#4,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	S,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
2:				
or.l	#cdr_listtag,D0	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
3:				
move.l	D0,(A1)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	D1,A1	4 (0)	20 (0)	28 (7)
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
4:				
decdr	(D0,S,Lcv0,Lco0,1)			
Lcv0:		5 (0)	28 (0)	35 (6)
move.l	(A1)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bmi	6	1/3	4/6	5/9
5:				
subq.l	#4,A1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	A1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
movea.l	S,A1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	2	3	6	9
6:				
decdr	(D1,A1,Lcv1,Lco1,Lnc1)			
Lcv1:		5 (0)	28 (0)	35 (7)
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bgt	7	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
7:				
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	3	3	6	9

unify_list (cont.)				
Instruction		**best**	**cache**	**worst**
Lncl:		2 (0)	8 (0)	12 (4)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
exg	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	2	3	6	9
Lco1:		6 (0)	14 (0)	17 (3)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	3	3	6	9
Lco0:		6 (0)	14 (0)	17 (3)
move.l	(A1),D1	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lfail	1/3	4/6	5/9
decdr	(D1,A1,Lcv2,Lco2,Lfail)			
Lcv2:		5 (0)	28 (0)	35 (6)
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	3	3	6	9
Lco2:		6 (0)	14 (0)	17 (3)
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	_unify_cdr	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
_unify_cdr:				
test_struct	(D0, Lfail)	1/3	16/18	23/27
test_struct	(D1, Lfail)	1/3	16/18	23/27
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	unify_struct	3	6	9
_loop:				
dereference	(D0, Lv, Lnv)			
Lv:		5 (0)	12 (0)	17 (5)
dereference	(D1, Lvv, Lvnv)			
Lvnv:		6 (0)	12 (0)	16 (3)
movea.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	01	1/3	4/6	5/9
exg	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	D1,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
move.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	04	3	6	9

unify_list (cont.)

Instruction		best	cache	worst
Lvv:		5 (0)	12 (0)	17 (5)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	10	1/3	4/6	5/9
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	20	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	02	1/3	4/6	5/9
01:				
movea.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
02:				
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	6 (0/1/1)
trail	D1,A0	4+ t (0)	20+ t (0)	28+ t (7)
bra	_continue	3	6	9
10:				
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	15	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	02	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
11:				
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	02	1/3	4/6	5/9
15:				
cmp.l	D0,D1	0 (0/0/0)	4 (0/0/0)	5 (0/1/0)
blt	11	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	11	3	6	9
20:				
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
ble	11	1/3	4/6	5/9
bra	01	3	6	9
Lnv:				
dereference	(D1,Lnvv,Lnn)			
Lnvv:		5 (0)	12 (0)	17 (5)
btst	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	03	1/3	4/6	5/9
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	03	3	6	9

unify_list (cont.)

Instruction		best	cache	worst
Lnn:		6 (0)	12 (0)	16 (3)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bclr	#cdr,D1 1 (0/0/0)	4 (0/0/0)	5 (0/1/0)	
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
beq	_continue	1/3	4/6	5/9
switch_tag	(D0, Lfail, Ll, Lfail, Ls)	8	27	36
Ll:				
test_list	(D1, Lfail)	6/4	12/10	16/12
lea	_unify_list,A0	2+ 2 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
move.l	D0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	D1,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	A0, -(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
bra	_continue	3	6	9
Ls:				
test_struct	(D1, Lfail)	3/1	18/16	27/23
lea	_unify_struct,A0	2+ 2 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
move.l	D0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	D1,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	A0, -(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
bra	_continue	3	6	9
Lfail:				
move.l	pdlBase(MP),PDL	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
bra	_fail	3	6	9
cycles	entry			
	_unify_list	19 (4)	42 (4)	55 (16)
	unify_list	11 (2)	30 (2)	41 (12)
	per elmement			
	var,var	46+ d+ t	(4)	108+ d+ t (4)
	var,nvar	38+ d+ t (3)	99+ d+ t (3)	131+ d+ t (32)
	nvar,var	39+ d+ t (3)	94+ d+ t (3)	127+ d+ t (38)
	const,const	28+ d+ c (2)	58+ d+ c (2)	70+ d+ c (16)
	list,list	53+ d+ ul (6)	114+ d+ ul (6)	149+ d+ ul (32)
	struc,struc	50+ d+ us (6)	120+ d+ us (6)	160+ d+ us (37)
	exit			
	const,const	30+ d+ c (1)	64+ d+ c (1)	80+ d+ c (15)
	c_var,ncdr	31+ d+ c (3)	92+ d+ c (3)	122+ d+ c (28)
	ncdr,c_var	19+ d+ c (2)	70+ d+ c (2)	90+ d+ c (22)
	var,var	41+ d+ c+ t (3)	126+ d+ c+ t (3)	164+ d+ c+ t (37)
	struc,struc	35+ d+ us (3)	84d+ us (3)	113+ d+ us (31)

d time needed for dereference   t time needed for trail
c time needed for decdr
ul and us are the time needed to unify a list or a structure respectively

unify_struct

input:	D0, D1
output:	
temps:	D0,D1,S,A1,A0,PDL
function:	general unification routine for struct

Instruction		**best**	**cache**	**worst**
_unify_struct:				
move.l	(PDL)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
move.l	(PDL)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
unify_struct:				
and.b	#xfc,D1	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
and.b	#xfc,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
movea	D0,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
movea	D1,A1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
cmp.l	(A1)+ ,D0	0+ 4 (1/0/0)	2+ 4 (1/0/0)	3+ 4 (1/1/0)
beq	loop	1/3	4/6	5/9
bra	Lfail	3	6	9
_continue:				
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bmi	50	1/3	4/6	5/9
move.l	(A1)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bmi	Lfail	1/3	4/6	5/9
loop:				
dereference	(D0, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
dereference	(D1, Lrr, Lrn)			
Lrn:				
move.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	1	1/3	4/6	5/9
exg	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D1,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	3	3	6	9
Lrr:		5 (0)	12 (0)	17 (5)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	10	1/3	4/6	5/9
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	20	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	2	1/3	4/6	5/9
1:				
movea.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
2:				
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
3:				
trail	D1,A0	4+ t (0)	20+ t (0)	28+ t (7)
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)

unify_struct (cont.)				
Instruction		best	cache	worst
10:				
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bmi	15	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
blt	2	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
11:				
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	2	1/3	4/6	5/9
15:				
cmp.l	D0,D1	0 (0/0/0)	4 (0/0/0)	5 (0/1/0)
blt	11	1/3	4/6	5/9
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	11	3	6	9
20:				
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
ble	11	1/3	4/6	5/9
bra	1	3	6	9
Lnref:		6 (0)	12 (0)	16 (0)
dereference	(D1, Lnr, Lnn)			
Lnr:		5 (0)	12 (0)	17 (5)
btst	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
beq	2	3	6	9
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	3	3	6	9
Lnn:		6 (0)	12 (0)	16 (3)
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
beq	_continue	1/3	4/6	5/9
switch_tag	(D0, Lfail, Ll, Lfail, Ls)	4	18	25
Ll:				
test_list	(D1, Lfail)	4/6	10/12	12/16
lea	_unify_list,A0	2+ 2 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
move.l	D0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	D1,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	A0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
bra	_continue	3	6	9
Ls:				
test_struct	(D1, Lfail)	1/3	16/18	23/27
lea	_unify_struct,A0	2+ 2 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
move.l	D0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	D1,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	A0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
bra	_continue	3	6	9

unify_struct (cont.)

Instruction		best	cache	worst
50:				
move.l	(A1),D1	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lfail	1/3	4/6	5/9
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	Lfail	1/3	4/6	5/9
cmp.l	#cdr_nil,D0	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bne	Lfail	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
Lfail:				
move.l	pdlBase(MP),PDL	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
bra	_fail	3	6	9
cycles	entry			
	_unify_struct	19 (4)	42 (4)	55 (16)
	unify_struct	11 (2)	30 (2)	41 (12)
	per element			
	var,var	46+ d+ t (4)	108+ d+ t (4)	150+ d+ t (42)
	nvar,var	38+ d+ t (3)	99+ d+ t (3)	131+ d+ t (32)
	var,nvar	39+ d+ t (3)	94+ d+ t (3)	127+ d+ t (38)
	const,const	25+ d (2)	52+ d (2)	68+ d (16)
	list,list	47+ d+ ul (5)	103+ d+ ul (5)	134+ d+ ul (25)
	struc,struc	44+ d+ us (5)	109+ d+ us (5)	145+ d+ us (25)
	exit	17 (2)	38 (2)	49 (13)

d time needed for dereference
t time needed for trail
ul and us time needed to unify a list or a structure respectively

A.5. Read Unification Routines

unify_variable_X				
input:	Xi, S			
output:				
temp:				
function:	read mode unification when Xi in register			
Instruction		best	cache	worst
r_unify_variableX:	Xi	in	register	
move.l	(S)+,Xi	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(Xi,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,Xi	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	Xi,S	4 (0)	20 (0)	28 (7)
move.l	D1,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
read_unify_				
cycles	base case	7 (1)	12 (1)	16 (4)
	invoke decdr	7+ c (1)	18+ c (1)	24+ c (7)
	change to w mode	25+ c+ t (3)	86+ c+ t (3)	113+ c+ t (27)
c time needed to decdr				
t time needed to trail				

unify_variable_X				
input:	Xi in memory			
output:				
function:	unify in read-mode when Xi in the memory			
Instruction		best	cache	worst
unify_variable_X:				
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(D0,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	D0,S	4 (0)	20 (0)	28 (7)
move.l	D1,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
move.l	D0,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
read_unify_				
cycles	base case	10 (2)	17 (2)	23 (6)
	decdr	10+ c (2)	23+ c (2)	31+ c (9)
	change to w mode	28+ c+ t (4)	89+ c+ t (4)	117+ c+ t (29)
c time needed to decdr				
t time needed to trail				

unify_variable_Y				
input:	Yi			
output:				
function:	unify in read mode			
Instruction		best	cache	worst
r_unify_variableY:				
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(D0,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	D0,S	4 (0)	20 (0)	28 (7)
move.l	D0,-4•i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
move.l	D0,-4•i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
read_unify_				
cycles	base case	10 (2)	17 (2)	23 (6)
	decdr	10+ c (2)	23+ c (2)	31+ c (9)
	change to w mode	28+ c+ t (4)	89+ c+ t (4)	117+ c+ t (29)
c time needed to decdr				
t time needed to trail				

unify_value_X				
input:	Xi in register			
output:				
function:	unify in read mode			
Instruction		best	cache	worst
r_unify_valueX:				
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(D0,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	D0,S	4 (0)	20 (0)	28 (7)
dereference	(Xi,Lref,Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,Xi	0+ 3 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bgt	1	1/3	4/6	5/9
move.l	H,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	Xi,A0	4 (0)	20 (0)	28 (7)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
1:				
move.l	Xi,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lnref:		6 (0)	12 (0)	16 (3)
move.l	Xi,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
move.l	Xi,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
brs	unify	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
read_unify_				
cycles	base case	15+ u (3)	26+ u (3)	39+ u (10)
	decdr	15+ c+ u (3)	32+ c+ u (3)	47+ c+ u (13)
	change to w mode	25+ c+ t (3)	84+ c+ t (3)	110+ c+ t (26)
c time needed to decdr				
t time needed to trail				
u time needed to unify				

unify_value_Y

input:	Yi
output:	
function:	unify Yi in read mode

Instruction		best	cache	worst
r_unify_valueY:				
move.l	-4•i(E),D1	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(D0,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
movea.l	D1,A1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	D0,S	4 (0)	20 (0)	28 (7)
move.l	A1,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
dereference	(D1,Lref,Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,D1	0+ 3 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bgt	5	1/3	4/6	5/9
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
movea.l	D1,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	2	1/3	4/6	5/9
move.l	H,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/0/1)
1:				
trail	D1,A0	4 (0)	20 (0)	28 (7)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
2:				
move.l	H,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	1	3	6	9
5:				
Lnref:		6 (0)	12 (0)	16 (0)
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D1,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	_unify	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
read_unify_				
cycles	base	18+ u (4)	31+ u (4)	45 (12)
	decdr	18+ c+ u (4)	37+ c+ u (4)	53+ c+ u (15)
	change to w mode	35+ c+ t (4)	111+ c+ t (4)	146+ c+ t (35)
	no move	40+ c+ d+ t (4)	123+ c+ d+ t (4)	164+ c+ d+ t (41)
	move	53+ c+ d+ t (5)	169+ c+ d+ t (5)	227+ c+ d+ t (57)

c time needed to decdr
d time needed to dereference
t time needed to trail
u time needed to unify

unify_constant

input: Constant C
output:
function: unify with C

Instruction		best	cache	worst
r_unify_constantC:				
move.l	(S)+,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(D0,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	D0,S	4 (0)	20 (0)	28 (7)
move.l	#C,(H)+	4 (0/0/1)	8 (0/0/1)	7 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
move.l	#C,D1	0+ 0 (0/0/0)	6 (0/0/0)	5 (0/1/0)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	unify1	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
read_unify_				
cycles	base case	15+ u (3)	30+ u (3)	41+ u (10)
	decdr	15+ c+ u (3)	36+ c+ u (3)	49+ c+ u (13)
	change to w mode	25+ c+ t (3)	88+ c+ t (3)	112+ c+ t (26)

c time needed to decdr
t time needed to trail
u time needed to unify

unify_nil

input:
output:
function: unify NIL in read mode

instruction		best	cache	worst
r_unify_nil:				
move.l	(S),D0	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	_fail	1/3	4/6	5/9
decdr	(D0,S,Lcr,Lcnr,_fail)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	#cdr_nil,(S)	3 (0/0/1)	8 (0/0/1)	7 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
trail	D0,S	4 (0)	20 (0)	28 (7)
bra	_continue	3	6	9
Lcnr:		6 (0)	14 (0)	17 (3)
cmp.l	#cdr_nil,D0	0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
beq	_continue	1/3	4/6	5/9
bra	_fail	3	6	9
cycles	cdr nref	13+ c (1)	36+ c (1)	46+ c (11)
	cdr ref	20+ c+ t (2)	76+ c+ t (2)	96+ c+ t (21)

c time needed to decdr
t time needed to trail

unify_cdr				
input:	Xi			
output:				
function:	Xi is unified with the Cdr of a list			
Instruction		**best**	**cache**	**worst**
r_unify_cdrX:				
move.l	(S),D0	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	1	1/3	4/6	5/9
decdr	(D0,S,Lcr,Lcrn,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
Lcrn:		6 (0) •	14 (0) •	17 (3) •
move.l	D0,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	_continue	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
subq	#4,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
1:				
move.l	S,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,Xi	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bra	_continue	3	6	9
cycles	base case	9 (1)	26 (1)	36 (9)
	decdr to cnref	13+ c (1)	32+ c (1)	41+ c (9)
	decdr to cref	13+ c (1)	50+ c (1)	64+ c (13)
	decdr to others	9+ c (1)	34+ c (1)	47+ c (13)
	Xi in memory	3 (1) more	3 (1) more	4 (1) more
c time needed to decdr				

unify_cdr_Y				
input:	Yi			
output:				
function:	unify Yi with the cdr of a list			
Instruction		**best**	**cache**	**worst**
r_unify_cdrY:				
move.l	(S),D0	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	1	1/3	4/6	5/9
decdr	(D0,S,Lcr,Lcrn,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
Lcrn:		6 (0)	14 (0)	17 (3)
move.l	D0,-4•i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
bra	_continue	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
subq	#4,S	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
1:				
move.l	S,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D0	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D0,-4•i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
bra	_continue			
cycles	base case	12 (2)	31 (2)	43 (11)
	decdr to cnref	16+ c (2)	37+ c (2)	48+ c (11)
	decdr to cref	16+ c (2)	55+ c (2)	71+ c (15)
	decdr to others	12 (2)	39 (2)	54 (15)
c time needed to decdr				

unify_void				
input:				
output:				
function:	unify unnamed variable in read mode			
Instruction		**best**	**cache**	**worst**
r_unify_void:				
move.l	(S)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
bge	Lo	1/3	4/6	5/9
decdr	(D0,S,Lcr,_fail,Lo)			
Lcr:		5 (0)	28 (0)	35 (6)
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
or.l	#cdr_listtag,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
move.l	D1,(S)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bset	#cdr,D0	1 (0/0/0)	4 (0/0/1)	5 (0/1/1)
trail	D0,S	4 (0)	20 (0)	28 (7)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
Lo:		2 (0)	8 (0)	12 (4)
read_unify_				
cycles	base case	7 (1)	12 (1)	16 (4)
	decdr	7+ c (1)	18+ c (1)	24+ c (7)
	change to w mode	25+ c+ t (3)	84+ c+ t (3)	101+ c+ t (27)
c time needed to decdr				
t time needed to trail				

A.6. Write Unification Routines

unify_variable				
input:	Xi			
output				
function:	unify Xi in write mode, Xi in register			
Instruction		best	cache	worst
w_unify_variableX:				
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	Xi,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
write_unify_				
cycles	Xi in register	4 (1)	6 (1)	8 (3)

unify_variable_X				
input: Xi				
output:				
function:	unify Xi in write mode when Xi is in memory			
Instruction		best	cache	worst
w_unify_variableX:				
move.l	H,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	D0,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
move.l	D0,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
write_unify_				
cycles		7 (2)	11 (2)	15 (5)

unify_variable_Y				
input: Yi				
output:				
function:	unify Yi in write mode			
Instruction		best	cache	worst
w_unify_variableY:				
move.l	H,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	D0,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
move.l	D0,-4•i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
write_unify_				

unify_value_X				
input: Xi in register				
output:				
function:	unify Xi. Need to dereference to handle unsafe value			
Instruction		best	cache	worst
cycles		7 (2)	11 (2)	15 (5)
w_unify_value_X:				
dereference	(Xi, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,Xi	0+0 (1/0/0)	2+4 (1/0/0)	3+5 (1/2/1)
bge	1	1/3	4/6	5/9
move.l	H,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	Xi,A0	4 (0)	20 (0)	28 (7)
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
1:				
move.l	Xi,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	unify_write_	3	6	9
Lnref:		6 (0)	12 (0)	16 (3)
move.l	Xi,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
write_unify_				
cycles	Xi nref	10+ d (1)	16+ d (1)	21+ d (5)
	ref nmove	16+ d (1)	40+ d (1)	56+ d (15)
	ref move	20+ d+ t (2)	58+ d+ t (2)	80+ d+ t (22)
d time to dereference				
t time to trail				

unify_value_X

input:	Xi in memory
output:	
function:	unify Xi in write mode

Instruction		best	cache	worst
w_unify_valueX:				
move.l	Xi(MP),D1	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
dereference	(D1, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,D1	0+ 0 (1/0/0)	2+ 4 (1/0/0)	3+ 5 (1/2/0)
bgt	1	1/3	4/6	5/9
move.l	H,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	D1,A0	4 (0)	20 (0)	28 (7)
move.l	H,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify	3	6	9
1:				
move.l	D1,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
bclr	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bra	2	3	6	9
Lnref:		6 (0)	12 (0)	16 (3)
move.l	D1,Xi(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
2:				
move.l	D1,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
write_unify_				
cycles	nref	16+ d (3)	28+ d (3)	37+ d (9)
	ref nmove	22+ d (3)	52+ d (3)	70+ d (19)
	ref move	26+ d+ t (4)	70+ d+ t (8)	94+ d+ t (26)
d time to derefence				
t time to trail				

unify_value_Y				
input:	Yi			
output:				
function:	unify Yi in write mode			
Instruction		**best**	**cache**	**worst**
w_unify_valueY:				
move.l	-4*i(E),D0	3 (1/0/0)	7 (1/0/0)	9 (1/2/0)
derefence	(D0, Lref, Lnref)			
Lref:		5 (0)	12 (0)	17 (5)
cmp.l	#stack_base,D0	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bge	1	1/3	4/6	5/9
move.l	H,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	D0,A0	4 (0)	20 (0)	28 (7)
move.l	H,-4*i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	write_unify_	3	6	9
1:				
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D0,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
bra	unify_write_	3	6	9
Lnref:		6 (0)	12 (0)	16 (3)
move.l	D0,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
write_unify_				
cycles	ref move	26+ d (4)	68+ d (4)	93+ d (24)
	ref no move	19+ d (2)	55+ d (2)	62+ d (13)
	nref	13+ d (1)	23+ d (1)	30+ d (5)
d time for dereference				

unify_constant				
input:	Constant C			
output:				
function:	unify C in write mode			
Instruction		**best**	**cache**	**worst**
w_unify_constantC:				
move.l	#C,(H)+	4 (0/0/1)	8 (0/0/1)	7 (0/1/1)
write_unify_				
cycles		0 (1)	8 (1)	7 (2)

unify_cdr_X				
input:	Xi			
output:				
function:	unify Xi with the Cdr of a list in write mode			
Instruction		**best**	**cache**	**worst**
w_unify_cdr_X:				
move.l	H,Xi	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,Xi	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	Xi,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
cycles	Xi in register	5 (1)	10 (1)	13 (4)
	Xi in memory	8 (2)	15 (2)	20 (6)

unify_cdr_Y				
input:	Yi			
output:				
function:	unify Yi with the cdr of a list in write mode			
Instruction		best	cache	worst
w_unify_cdrY:				
move.l	H,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
move.l	D1,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
move.l	D1,-4•i(E)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
cycles		8 (2)	15 (2)	20 (6)
w_unify_nil:				
move.l	#cdr_nil,(H)+	4 (0/0/1)	8 (0/0/1)	7 (0/1/1)
cycles		4 (1)	8 (1)	7 (2)

unify_void				
input:				
output:				
function:	create unnamed variable on the heap			
Instruction		best	cache	worst
w_unify_void:				
move.l	H,(H)+	4 (0/0/1)	4 (0/0/1)	5 (0/1/1)
_unify_write_				
cycles		4 (1)	4 (1)	5 (2)

A.7. Escape Routines

escape comparison

input:	D0,D1
output:	
temps: D0, D1	
funciton:	basic expansion for all comparisions

Instruction		best	cache	worst
escape_cmp/2:				
dereference	(D0, Lfail, Lnref)			
Lnref:		4 (0)	12 (0)	16 (3)
switch_tags	(D0, Lfail, Lls, Lc, Lls)	4 (0)	18 (0)	25
Lc:				
btst	#2,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bne	Lfail	1/3	4/6	5/9
bra	_continue	3	6	9
Lls:				
move.l	D1,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
brs	eval	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
move.l	(PDL)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
_continue:				
dereference	(D1,Lfail,Lnref1)			
Lnref1:		4	10	12
switch_tags	(D1,Lls1,Lfail,Lc1,Lls1)	4	18	25
Lc1:				
cmp.l	D1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bncc	Lfail	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
Lls1:				
move.l	D0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
exg	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
brs	eval	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
cmp.l	(PDL)+ ,D1	0+ 4 (1/0/0)	2+ 4 (1/0/0)	3+ 4 (1/1/0)
bncc	Lfail	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
Lfail:				
move.l	pdlBase(MP),PDL	3 (1/0/0)	7 (1/0/0)	9 (1/1/0)
bra	_fail	3	6	9
cycle	const,const	31+ d	86+ d	113+ d
	struc,const	38+ d+ e	90+ d+ e	120+ d+ e
	const,struct	43+ d+ e	104+ d+ e	139+ d+ e
	struct,struct	50+ d+ e	108+ d+ e	146+ d+ e

d time needed to dereference
e time needed to evaluate the expressions

escape_is/2 (D0, D1)

input:	D0, D1
output:	
function:	evaluate D0. If D1 is reference type, assign it the result else compare it with the result.

Instruction		best	cache	worst
escape_is/2:				
dereference	(D0, Lfail, Lnref)			
Lnref:		4	10	12
switch_tag	(D0, Lls, Lfail, Lc, Lls)	4	18	25
Lc:				
btst	#2,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bne	Lfail	1/3	4/6	5/9
bra	_continue	3	6	9
Lls:				
move.l	D1,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
brs	eval	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
move.l	(PDL)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
_continue:				
dereference	(D1, Lref1, Lnref1)			
Lref1:		4	14	17
bset	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
btst	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bne	1	1/3	4/6	5/9
bclr	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
1:				
move.l	D0,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	D1,A0	4	20	28
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
Lnref1:		4	10	12
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	Lfail	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
cycles	const,const	27+ d	68+ d	88+ d
	const,struct	34+ d+ e	72+ d+ e	95+ d+ e
	var,const	51+ d	104+ d	137+ d
	var,struct	58+ d	108+ d+ e	144+ d+ e

d time to dereference
e time to evaluate the expression

escape_is/4

input:	X0,X1,X2,X3
output:	If X0 is reference, it's bound to X1 op X3 if X0 is a number, it's compared with X1 op X3 otherwise, fail
temp:	A0,D0,D1
function:	This is a local optimization. X1 and X3 are constants. X2 is the oprand aligned for Jumptable.

Instruction		best	cache	worst
move.l	X1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
dereference	(D0,Lfail,Lnref)	4	10	12
Lnref:				
lsr.l	#1,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
asr.l	#4,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
move.l	D0,X1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	X3,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
dereference	(D0,Lfail,Lnref1)	4	10	14
Lnref1:				
lsr.l	#1,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
asr.l	#4,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
move.l	D0,X3	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	X2,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
dereference	(D0,Lfail,Lnref2)	4	10	12
Lnref2:				
cmp.b	#0x14,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
blt	Lfail	1/3	4/6	5/9
jmp	(Table,PC,D1.w)	1+ 3 (0/0/0)	4+ 6 (0/0/0)	7+ 6 (0/2/0)
Table:				
add:	ADD1			
sub:	SUB1			
mul:	MUL1			
div:	DIV1			
mod:	MOD1			
ADD1:				
move.l	X1,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
add.l	X3,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
SUB1:				
move.l	X1,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
sub.l	X3,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)

escape_is/4 (cont.)				
instruction		best	cache	worst
MUL1:				
move.l	X1,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
muls.w	X3,D1	25 (0/0/0)	27 (0/0/0)	28 (0/1/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
DIV1:				
move.l	X1,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
divs.l	X3,D1	54 (0/0/0)	56 (0/0/0)	56 (0/1/0)
and.l	#quotmask,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
MOD1:				
move.l	X1,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
divs.l	X3,D1	54 (0/0/0)	56 (0/0/0)	56 (0/1/0)
move.l	#16,D0	0 (0/0/0)	6 (0/0/0)	5 (0/1/0)
lsr.l	D0,D1	3 (0/0/0)	6 (0/0/0)	6 (0/1/0)
_end:				
lsl.l	#3,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
or.b	#2,D1	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
move.l	X0,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
dereference	(D0,Lref,Lnref)	deref/b	deref/c	deref/w
Lref:		4	14	17
bset	#cdr,D1	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
btst	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bne	2	1/3	4/6	5/9
bclr	#cdr,D1	1(0/0/0)	4 (0/0/0)	5 (0/1/0)
2:				
move.l	D1,(A0)	3 (0/0/1)	4 (0/0/1)	5 (0/1/1)
trail	D0,A0	4+ t	20+ t	28+ t
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
Lnref:				
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bne	_fail	1/3 (0/0/0)	4/6 (0/0/0)	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
cycles	des. var	47+ d+ t	148+ d+ t	186+ d+ t
	des. const	36+ d+ t	110+ d+ t	140+ d+ t
	add or sub	3	10	15
	mul	28	35	40
	div	57	70	76
	mod	57	70	70
d time to dereference				
t time to trail				

eval D0	
input:	D0
output:	D0
temps:	PDL, A0, A1, D0, D1
function:	evaluate D0 and put the result in D0
	This is a recursive subroutine

Instruction		best	cache	worst
eval:				
dereference	(D0, Lf, Lnref)	4	10	12
Lnref:				
switch_tag	(D0,Lf,Ll0,Lc0,Ls0)	4	18	25
Lc0:				
btst	#3,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
bne	Lf	1/3	4/6	5/9
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
Ll0:				
and.b	0xfc,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
movea.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(A0)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
move.l	(A0),D1	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
decdr	(D1,Lf,Lcnr,Lf)	9	12	16
Lcnr:				
cmp.l	#cdr_nil,D1	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
beq	eval	1/3	4/6	5/9
Lf:				
move.l	pdlBase(MP),PDL	3 (1/0/0)	7 (1/0/0)	9 (1/1/0)
bra	_fail	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
Ls0:				
and.b	0xfc,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
movea.l	D0,A0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(A0)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
and.w	#arity2,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
bne	Lf	1/3	4/6	5/9
move.l	(A0)+ ,-(PDL)	6 (1/0/1)	7 (1/0/1)	9 (1/1/1)
move.l	(A0)+ ,-(PDL)	6 (1/0/1)	7 (1/0/1)	9 (1/1/1)
move.l	(A0)+ ,D0	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
brs	eval	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
move.l	(PDL)+ ,D1	4 (1/0/0)	6 (1/0/0)	7 (1/1/0)
move.l	D0,-(PDL)	3 (0/0/1)	5 (0/0/1)	6 (0/1/1)
move.l	D1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
brs	eval	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
lsr.l	#3,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
movea.l	D0,A1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	(PDL)+ ,D0	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)

		eval (cont.)		
instruction		best	cache	worst
lsr.l	#3,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
move.l	(PDL)+ ,D1	3 (1/0/0)	6 (1/0/0)	7 (1/1/0)
move.l	#16,A0	0 (0/0/0)	6 (0/0/0)	5 (0/1/0)
lsr.l	A0,D1	3 (0/0/0)	6 (0/0/0)	6 (0/1/0)
jmp	(Table,PD,D1.b)	1+ 3 (0/0/0)	4+ 6 (0/0/0)	7+ 6 (0/2/0)
Table:				
add:	ADD			
sub:	SUB			
mul:	MUL			
div:	DIV			
mod:	MOD			
ADD:				
add.l	A1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
SUB1:				
sub.l	A1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
MUL1:				
muls.w	A1,D0	25 (0/0/0)	27 (0/0/0)	28 (0/1/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
DIV1:				
divs.l	A1,D0	54 (0/0/0)	56 (0/0/0)	56 (0/1/0)
and.l	#quotmask,D0	0+ 0 (0/0/0)	2+ 4 (0/0/0)	3+ 5 (0/2/0)
bra	_end	3 (0/0/0)	6 (0/0/0)	9 (0/2/0)
MOD1:				
divs.l	A1,D0	54 (0/0/0)	56 (0/0/0)	56 (0/1/0)
move.l	#16,D1	0 (0/0/0)	6 (0/0/0)	5 (0/1/0)
lsr.l	D1,D0	3 (0/0/0)	6 (0/0/0)	6 (0/1/0)
_end:				
lsl.l	#3,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
or.b	#2,D0	0+ 0 (0/0/0)	2+ 2 (0/0/0)	3+ 3 (0/2/0)
rts		9 (1/0/0)	10 (1/0/0)	12 (1/2/0)
cycles	const	19+ d	46+ d	59+ d
	list	27+ c+ d+ e	70+ c+ d+ e	93+ c+ d+ e
	struc	71+ d+ e	157+ d+ e	202+ d+ e
	add, sub	3	8	12
	mul	28	33	37
	div	57	68	73
	mod	57	68	67
c time to decdr for lists				
d time to dereference				
t time to trail				

add, sub, mul

input:	X1, X2 - input argument X3 - result
output:	
function:	X1 and X2 can only be integers, and X3 may be a variable or an integer.

Instruction		best	cache	worst
add:				
dereference	(X1,_fail,Lnref)			
Lnref:		6 (0)	12 (0)	16 (3)
test_integer	(X1,_fail)	1/3	16/18	23/27
dereference	(X2,_fail,Lnref1)			
Lnref1:		6 (0)	12 (0)	16 (3)
test_integer	(X2,_fail)	1/3	16/18	23/27
move.l	X1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	X2,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
lsl.l	#1,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
lsl.l	#1,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
asr.l	#4,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
asr.l	#4,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
add.l	D1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
lsl.l	#3,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
or.b	#2,D0	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
move.l	X3,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	unify1	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
_continue:				
cycles	add	30+ d+ u-6 (2)	100+ d+ u-12 (2)	136+ d+ u-16 (36)
	sub	30+ d+ u-6 (2)	100+ d+ u-12 (2)	136+ d+ u-16 (36)
	mul.w	55+ d+ u-6 (2)	125+ d+ u-12 (2)	161+ d+ u-16 (36)

d time to dereference
u time to unify

div, mod	
input:	X1, X2 - input argument X3 - result
output:	
function:	X1 and X2 can only be integers, and X3 may be a variable or an integer.

Instruction		best	cache	worst
div:				
dereference	(X1,_fail,Lnref)			
Lnref:		6 (0)	12 (0)	16 (3)
test_integer	(X1,_fail)	1/3	16/18	23/27
dereference	(X2,_fail,Lnref1)			
Lnref1:		6 (0)	12 (0)	16 (3)
test_integer	(X2,_fail)	1/3	16/18	23/27
move.l	X1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	X2,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
lsl.l	#1,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
lsl.l	#1,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
asr.l	#4,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
asl.l	#4,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
div.w	D1,D0	54 (0/0/0)	56 (0/0/0)	56 (0/1/0)
swap	D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
clr.w	D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
swap	D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
ext.w	D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
lsl.l	#3,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
bcrl	#cdr,D0	1 (0/0/0)	4 (0/0/0)	5 (0/1/0)
or.b	#0x2,D0	0 (0/0/0)	4 (0/0/0)	6 (0/2/0)
move.l	X3,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	PDL,pdlBase(MP)	3 (0/0/1)	5 (0/0/1)	7 (0/1/1)
brs	unify1	5 (0/0/1)	7 (0/0/1)	13 (0/2/1)
_continue:				
cycles	div	89+ d+ u-6 (2)	176+ d+ u-12 (2)	213+ d+ u-16 (41)
	mod	88+ d+ u-6 (2)	172+ d+ u-12 (2)	209+ d+ u-16 (40)

d time to dereference
u time to unify

Compiler Optimized Comparisons

Input:	X1, X2 - both can only be integers or *dereferenced* to integers
output:	
function:	Test to see if the condition hold, proceed or fail accordingly

Instruction		best	cache	worst
comp:				
dereference	(X1,_fail,Lnref)			
Lnref:		6 (0)	12 (0)	16 (3)
test_integer	(X1,_fail)	1/3	16/18	23/27
dereference	(X2,_fail,Lnref1)			
Lnref1:		6 (0)	12 (0)	16 (3)
test_integer	(X2,_fail)	1/3	16/18	23/27
move.l	X1,D0	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
move.l	X2,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
lsl.l	#1,D0	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
lsl.l	#1,D1	1 (0/0/0)	4 (0/0/0)	4 (0/1/0)
cmp.l	D0,D1	0 (0/0/0)	2 (0/0/0)	3 (0/1/0)
bncc	_fail	1/3	4/6	5/9
cycles	true	17 (0)	76 (0)	95 (24)
	false	19 (0)	78 (0)	99 (25)

Appendix B. Benchmark Results.

Table 1. Cycle Count Ratio (MC68020 vs. VLSI-PLM)

prog name	plm	68020 (best)	68020 (cache)	68020 (worst)
chat (Mulder/Tick)	1.0 †	2.43	5.71	7.71
ucb (Mulder/Tick)	1.0	2.14	5.00	6.42
boyer (Mulder/Tick)	1.0	1.86	4.43	5.71
chat (Patt/Chen) •	1.0	3.13	6.67	8.33
boyer (Patt/Chen) •	1.0	2.39	5.22	6.82
browse	1.0	3.00	6.23	8.55
con1	1.0	2.71	6.94	9.16
con6	1.0	3.60	7.91	10.35
differen	1.0	2.68	6.11	8.05
hanoi	1.0	2.79	5.70	7.50
mumath	1.0	3.13	6.75	8.74
nrev1	1.0	2.71	7.01	9.38
palin	1.0	3.09	7.23	9.40
pri2	1.0 ‡	2.29	3.03	3.49
qs4	1.0	2.32	5.09	6.73
queens	1.0	3.30	6.99	9.10
query	1.0 ‡	1.20	2.01	3.95

† Timing as reported in [1] is actually for their *plm1* model, which had a fixed size choice point frame and a single cycle memory.

• Cycle count ratios for chat and boyer are not comparable to those obtained in [1] since their numbers are for their plm model while ours are for the most up-to-date version of the VLSI-PLM.

‡ The cycle counts for the PLM are calculated with the assumption that an internally coupled MC68020 is used to evaluate the *escapes*. Therefore, cycle count is different for each of the three cases, depending on how the MC68020 evaluates the escapes, i.e., whether best, cache, or worst case applies.

Table 2. Relative Performance of Projected MC68020s vs. 10 MHz VLSI-PLM ‡

Program	PLM	MC68020				
	10 MHz	10 MHz	16.7 MHz	25 MHz	30 MHz	40 Mhz
con1 (best)		0.37	0.62	0.93	1.1	1.24
con1 (cache)	1.0	0.14	0.24	0.36	0.43	0.57
con1 (worst)		0.11	0.18	0.27	0.33	0.44
con6 (best)		0.29	0.48	0.72	0.87	1.16
con6 (cache)	1.0	0.13	0.22	0.33	0.39	0.52
con6 (worst)		0.10	0.17	0.25	0.30	0.40
differen (best)		0.37	0.61	0.92	1.10	1.47
differen (cache)	1.0	0.16	0.27	0.40	0.48	0.64
differen (worst)		0.12	0.20	0.30	0.37	0.49
hanoi (best)		0.35	0.59	0.89	1.06	1.41
hanoi (cache)	1.0	0.17	0.29	0.43	0.52	0.69
hanoi (worst)		0.13	0.22	0.33	0.39	0.52
mumath (best)		0.32	0.53	0.80	0.96	1.28
mumath (cache)	1.0	0.15	0.25	0.37	0.44	0.59
mumath (worst)		0.11	0.19	0.29	0.34	0.45
nrev1 (best)		0.37	0.62	0.92	1.11	1.48
nrev1 (cache)	1.0	0.14	0.24	0.36	0.43	0.57
nrev1 (worst)		0.11	0.18	0.27	0.32	0.43
palin (best)		0.34	0.57	0.85	1.02	1.36
palin (cache)	1.0	0.15	0.24	0.36	0.44	0.59
palin (worst)		0.11	0.19	0.28	0.34	0.45
pri2 (best)		0.44	0.60	0.81	0.92	1.17
pri2 (cache)	1.0 †	0.33	0.41	0.52	0.58	0.71
pri2 (worst)		0.29	0.35	0.44	0.49	0.59
qs4 (best)		0.43	0.72	1.07	1.29	1.72
qs4 (cache)	1.0	0.20	0.33	0.49	0.59	0.79
qs4 (worst)		0.15	0.25	0.37	0.45	0.59
queens (best)		0.31	0.51	0.77	0.92	1.23
queens (cache)	1.0	0.15	0.24	0.36	0.44	0.59
queens (worst)		0.11	0.19	0.28	0.33	0.44
query (best)		0.59	0.89	1.27	1.49	1.94
query (cache)	1.0 †	0.30	0.43	0.59	0.68	0.87
query (worst)		0.24	0.34	0.46	0.54	0.69

† The PLM timing for the escapes are calculated under the assumption of an internally coupled MC68020 running at the corresponding clock rate

‡ Relative performance is computed as the ratio of simulated VLSI-PLM execution time divided by the projected execution time of the processor indicated in that column.

Table 3. Relative Performance of Projected MC68020s vs. 16.7 MHz VLSI-PLM ‡

Program	PLM	MC68020			
	16.7 MHz	16.7 MHz	25 MHz	30 MHz	40 MHz
con1 (best)		0.37	0.56	0.67	0.89
con1 (cache)	1.0	0.14	0.22	0.26	0.35
con1 (worst)		0.11	0.16	0.20	0.27
con6 (best)		0.29	0.43	0.52	0.69
con6 (cache)	1.0	0.13	0.20	0.24	0.32
con6 (worst)		0.10	0.15	0.18	0.24
differen (best)		0.37	0.55	0.66	0.88
differen (cache)	1.0	0.16	0.24	0.28	0.37
differen (worst)		0.12	0.18	0.22	0.29
hanoi (best)		0.35	0.53	0.64	0.85
hanoi (cache)	1.0	0.17	0.26	0.31	0.41
hanoi (worst)		0.13	0.20	0.24	0.32
mumath (best)		0.32	0.48	0.57	0.76
mumath (cache)	1.0	0.15	0.22	0.27	0.36
mumath (worst)		0.11	0.17	0.21	0.28
nrev1 (best)		0.37	0.55	0.66	0.88
nrev1 (cache)	1.0	0.14	0.21	0.26	0.35
nrev1 (worst)		0.11	0.16	0.19	0.25
palin (best)		0.34	0.51	0.61	0.81
palin (cache)	1.0	0.15	0.22	0.26	0.35
palin (worst)		0.11	0.17	0.20	0.27
pri2 (best)		0.44	0.56	0.63	0.78
pri2 (cache)	1.0 †	0.33	0.39	0.43	0.51
pri2 (worst)		0.29	0.34	0.37	0.43
qs4 (best)		0.43	0.64	0.77	1.03
qs4 (cache)	1.0	0.20	0.29	0.35	0.47
qs4 (worst)		0.15	0.22	0.27	0.36
queens (best)		0.31	0.46	0.55	0.73
queens (cache)	1.0	0.15	0.22	0.26	0.35
queens (worst)		0.11	0.17	0.20	0.27
query (best)		0.59	0.82	0.95	1.22
query (cache)	1.0 †	0.30	0.39	0.45	0.57
query (worst)		0.24	0.31	0.35	0.45

† The PLM timing is calculated by assuming an internally coupled MC68020 running at the corresponding clock rate.

‡ Relative performance is computed as the ratio of simulated VLSI-PLM execution time divided by the projected execution time of the processor indicated in that column.

Table 4. Calculated vs. Real Cycles for MC68020

Program	Calculated			Real Time on Tower		on SUN 3/260
	Best	Cache	Worst	Macro Expand	Quintus	Macro Expand
con1	501	1280	1700	1570	1960	1460
con6	3030	6600	8640	8990	8400	8020
hanoi	139000	276000	361000	399000	367000	335000
nrev1	57100	148000	198000	196000	141000	181000
qs4	113000	256000	338000	357000	344000	309000
queens	109000	223000	289000	327000	232000	268000

Remarks:

Benchmark programs run on the Tower/32 suffer at least one wait state for each data access. On the other hand, the programs run on the SUN 3/260 can approach the ideal MC68020 speed if every access results in a cache hit.

There are optimizations in the Quintus system, as demonstrated by the results for nrev1 and queens.

Table 5. Code Size Comparison

Program	MC68020 Size	PLM Size	SUN/PLM	SUN/PLM †
con1	2388	74	32.3	4.7
con6	2388	79	30.2	4.7
hanoi	2100	120	17.7	3.8
nrev1	3608	269	13.7	5.3
qs4	5908	451	13.1	6.7
queens	9020	515	17.5	9.6
chat ‡	343130	21816	15.9	15.6
boyer ‡	271840	7615	36.6	33.8

† The PLM program size used in this column includes a run time support library of 429 bytes.

‡ The size for the MC68020 is based on estimation (with less than 2% of error).